# ENGR-UH 3511

# Computer Organization and Architecture

**Name:** Nishant Aswani

**Net ID:** nsa325

**Assignment Title:** Homework 3

# ENGR-UH 3511 Homework 3

Nishant Aswani (nsa325@nyu.edu)

September 15, 2019

## Question 2.31

We are given:

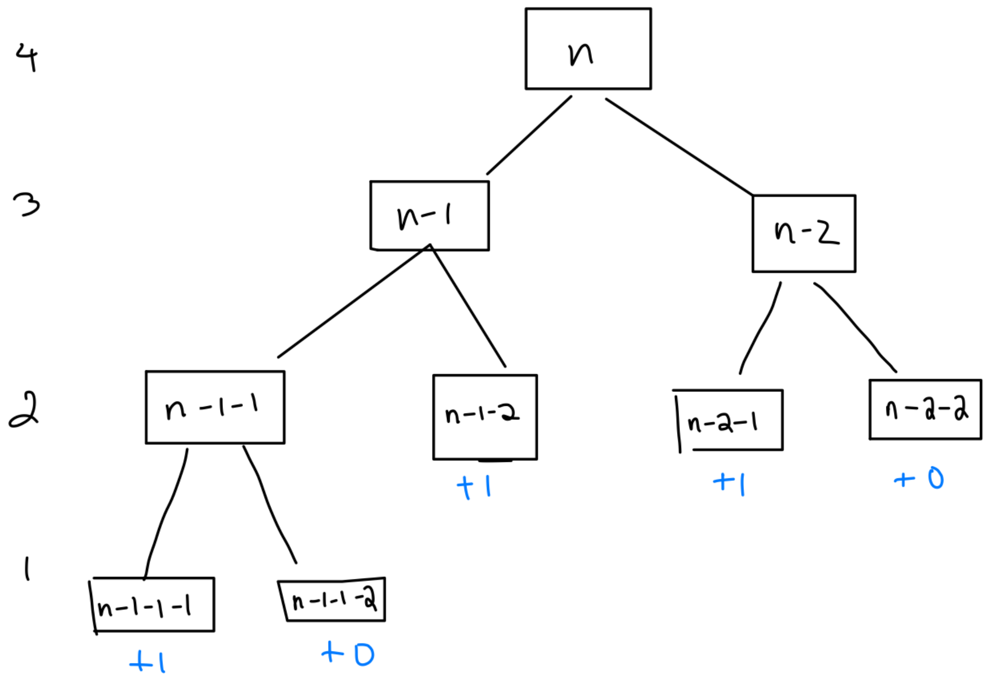
```
int fib(int n){
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

The C code above is a recursive implementation of the fibonacci sequence. The assembly implementation would then require the use of the `jal` instruction to grow a stack. As elements are popped off the stack, the program would sum up values to obtain the fibonacci number.

Below is the code, with comments:

```
.data
resultMessage: .asciiz "The fibonacci number is: "
fibIn: .byte 24             # fibIn
res: .byte 1                # result

.text
main:
  lb $a1, fibIn             # save fibIn into $a1
  li $a2, 1                 # load 1 into $a2
  beq $a2, $a1, edgeCase    # if fibIn == 1, jump to edgeCase
  jal fibRec                # jump to fibRec label and save return address
print:
  li $v0, 4                 # code for print_string
  la $a0, resultMessage     # point $a0 to resultMessage
  syscall                   # print the result
  li  $v0,1                 # code for print_int
  move  $a0, $v1            # put result in $a0
  syscall                   # print out the result
exit:
  li $v0, 10                # code for exit
  syscall                   # exit programs
fibRec:
  addi $sp, $sp, -12        # decrement stack pointer by 12 bytes
  sw $ra, 8($sp)            # save return address
  sw $a1, 4($sp)          # save fibIn
  sw $a2, 0($sp)          # save fibIn-1

  addi $a1, $a1, -1       # decrement the fibIn
  addi $a2, $a1, -1       # subtract the fibIn-1

  blt $a2, $zero, fibEnd  # if fibIn-1 == 0, start returning

  jal fibRec              # loop back to beginning

  li $t0, 1               # load 1 into $t0
check1:
  beq $t0, $a1, inc       # if $a1 == 1, increment final answer
check2:
  bne $t0, $a2, check3    # if $a2 != 1, go to check 3
  addi $v1, $v1, 1        # else increment
check3:
  bgt $a2, $t0, fibCall   # if $a2 > 1, jump to fibCall
  j fibEnd                # else continue returning
fibCall:
  addi $a1, $a2, -1       # $a1 = $a1 - 1
  addi $a2, $a2, -2       # $a2 = $a1 - 1
  j fibRec                # call fibRec again
fibEnd:
  lb $a2, 0($sp)          # store the fib-2 into $a2
  lb $a1, 4($sp)          # store the fib-1 into $a1
  lw $ra, 8($sp)          # store the return address
  addi $sp, $sp, 12       # increment the stack pointer 12 bytes
  jr $ra                  # jump to return address
inc:
  addi $v1, $v1, 1
  j check2
edgeCase:
  li $v1, 1               # loads 1 into result without recursing
  j print                 # jumps to print instruction
```

The recursive logic for n=4 is seen below:



**Not accounting for the print and exit instructions, this recursive implementation of the fib function in assembly is 30 instructions.**

## Question 2.39

We are given the binary value `0010 0000 0000 0001 0100 1001 0010 0100`. To carry this out, we will load the upper 16 bits, and then use `ori` to set the second half.

The decimal value for the 32 bit constant is 536955172. The first 16 bits lead to a decimal value 8193, while the second half are 18724.

```
.text
main:
  lui $t1, 8193                # 8193 = 0010000000000001
  ori $t1, $t1, 18724          # 18724 = 0100100100100100
exit:
  li $v0, 10                   # code for exit
  syscall                      # exit programs
```

We could also run the pseudoinstruction `"li $t0, 536955172"`, which would break down into the instructions above.

## Question 2.40

We are given `0x00000000` as the current PC address.

Jump instructions store the jump address in 26 bits. The address is then multiplied by 4, using a shift left logical by 2 operation. The multiplication by 4 actually helps stores 28 bit value within the jump address, because the last two bits are assumed to be zero.

Then, to make it 32 bits, the value is top concatenated with the most significant bits (MSB) of the PC to obtain the final jump address.

In this case, the MSB of the binary address are 0010, while the MSB of the PC are 0000. **Therefore, it would not be possible to use the jump instruction to make a single jump.**

## Question 2.41

We are given `0x00000600`, or , as the current PC address.

An I-type instruction is broken down as follows.

| opcode | rs | rt | immediate |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

Hence, only 16 bits are available to store the branch-to address. In order to resolve this, the 16-bit value is once again multiplied by 4 to effectively store an 18-bit value, which is then added to the next PC address (PC + 4).

The next PC address is `0x000620` or 1568. In order to jump to `0x20014924` or 536955172, we would have to be able to store

$\frac{536955172-1568}{4} = 134238401$.

However, 134238401, is a value that requires 28 bits. Thus, it cannot fit the 16 bits partition for an I-type instruction. **The branching would require more than one instruction.**

## Question 2.42

We are given that the current PC is 0x1FFFF000 and our goal is to get to address `0x20014924`. The next PC address is 0x1FFFF020

The difference, in decimal, between the next PC address and target address is $536955172-536866848 = 88324$. Dividing it by 4 gives us 22081.

In binary, this value is 0101011001000001, meaning it fits within 16 bits of the I-type instruction. **Hence, this branch can be carried out in a single instruction.**